NumLin: Linear Types for Linear Algebra

Dhruv C. Makwana[1 [orcidID]] and Neelakantan R. Krishnaswami[2 [orcidID]]

[1] dcm41@cam.ac.uk dhruvmakwana.com
[2] Department of Computer Science, University of Cambridge
nk480@cl.cam.ac.uk

Abstract. Briefly summarize the contents of the paper in 150–250 words.

Keywords: numerical, linear, algebra, types, permissions, OCaml

1 Introduction

NumLin is a functional programming language designed to express the APIs of low-level linear algebra libraries (such as BLAS/LAPACK) safely and explicitly. It does so by combining linear types, fractional permissions, runtime errors and recursion into a small, easily understandable, yet expressive set of core constructs. NumLin allows a novice to understand and work with complicated linear algebra library APIs; as well as point out subtle aliasing bugs and reduce memory usage in existing programs. In addition to this, NumLin's implementation supports several syntactic conveniences as well as a *usable* integration with real OCaml libraries.

1.1 Contributions

In this paper

- we describe NumLin, a linearly typed language for linear algebra programs
- we illustrate that NumLin's design and features are well-suited to its intended domain with progressively sophisticated examples
- we prove NumLin's soundness, using a step-indexed logical relation
- we describe a very simple, unification based type-inference algorithm for polymorphic fractional permissions (similar to ones used for parametric polymorphism), demonstrating an alternative approach to the dataflow analysis of Bierhof et al's *Fraction Polymorphic Permission Inference*
- we describe an implementation that is both compatible with and usable from existing code.

2 NumLin Overview and Examples

2.1 Overview

Linearity is at the heart of NumLin. Linearity allows us to express a pure-functional API for numerical library routines that mutate arrays and matrices. Linearity also restricts aliasing of (values which represent) pointers.

Intuitionism: ! and `Many` However, linearity by itself is not sufficient to produce an expressive enough programming language. For values such as booleans, integers, floating-point numbers as well as pure functions, we need to be able to use them *intuitionistically*, that is, more than once or not at all. For this reason, we have the ! constructor at the type level and its corresponding `Many` constructor and `let Many <id> = .. in ..` eliminator at the term level. Because we want to restrict how a programmer can alias pointers and prevent a programmer from ignoring them (a memory leak), NumLin enforces simple syntactic restrictions on which values can be wrapped up in a `Many` constructor (details in Section 3).

Fractional Permissions There are also valid cases in which we would want to alias pointers to a matrix. The most common is exemplified by the BLAS routine `gemm`, which (rather tersely) stands for *GEneric Matrix Multiplication*. A *simplified* definition of `gemm(`α`, A, B,` β`, C)` is $C := \alpha AB + \beta C$. In this case, `A` and `B` may alias each other but neither may alias `C`, because it is being written to. Related to *mutating* arrays and matrices is *freeing* them. Here, we would also wish to restrict aliasing so that we do not free one alias and then attempt to use another. Although linearity on its own suffices to prevent use-after-free errors when values are *not* aliased (a freed value is *out of scope* for the rest of the expression), we still need another simple, yet powerful concept to provide us with the extra expressivity of aliasing *without* losing any of the benefits of linearity.

Fractional permissions provide exactly this. Concretely, types of (pointers to) arrays and matrices are *parameterised* by a *fraction*. A fraction is either 1 (2^0) or exactly *half* of another fraction (2^{-k}, for natural k). The former represents complete ownership of that value: the programmer may mutate or free that value as they choose; the latter represents read-only access or a *borrow*: the programmer may read from the value but not write to or free it. Creating an array/matrix gives you ownership of it, so too does having one (with a fractional permission of 2^0) passed in as an argument.

In NumLin, we can produce two aliases of a single array/matrix, by *sharing* it. If the original alias had a fractional permission of 2^{-k} then the two new aliases of it will have a fractional permission of $2^{-(k+1)}$ each. Thanks to linearity, the original array/matrix with a fractional permission of 2^{-k} will be out of scope after the sharing. When an array/matrix is shared as such, we can prevent the programmer from freeing or mutating it by making the types of `free` and `set` (for mutation) require a *whole* (2^0) permission.

If we have two aliases *to the same matrix* with *identical* fractional permissions ($2^{-(k+1)}$), we can recombine or *unshare* them back into a single one, with a larger 2^{-k} permission. As before, thanks to linearity, the original two aliases will be out of scope after unsharing.

Runtime Errors Aside from out-of-bounds indexing, matrix unsharing is one of only *two* operations that can fail at runtime (the other being dimension checks,

```
let rec factorial ( !x : !int ) : !int =
    if x < 0 || x = 0 then
        1
    else
        x * factorial (x - 1) in factorial
;;
```

Fig. 1. Factorial function in NumLin.

such as for `gemm`). The check being performed is a simple sanity check that the two aliasing pointers passed to `unshare` point to the same array/matrix. Section 5 contains an overview of how we could remove the need for this by tracking pointer identities statically by augmenting the type system further.

Recursion The final feature of NumLin which makes it sufficiently expressive is recursion (and of course, conditional branches to ensure termination). Conditional branches are implemented by ensuring that both branches use the same set of linear values. A function can be recursive if it captures no linear values from its environment. Like with `Many`, this is enforced via simple syntactic restrictions on the definition of recursive functions.

2.2 Examples

Factorial Although a factorial function (Figure 1) may seem like an aggressively pedestrian first example, in a linearly typed language such as NumLin it represents the culmination of many features.

To simplifiy the design and implementation of NumLin's type system, recursive functions must have full type annotations (non-recursive functions need only their argument types annotated). Its body is a closed expression (with respect to the function's arguments), so it type-checks (since it does not capture any linear values from its environment).

The only argument is `!x : !int`. The ! annotation on `x` is a syntactic convenience for declaring the value to used intuitionistically, its full and precise meaning is described in Section 4.1.

The condition for an `if` may or may not use linear values (here, with `x < 0 || x = 0`, it does not). Any linear values used by the condition would not be in scope in either branch of the `if`-expression. Both branches use `x` differently: one ignores it completely and the other uses it twice.

All numeric and boolean literals are implicitly wrapped in a `Many` and all primitives involving them return a `!int`, `!bool` or `!elt` (types of elements of arrays/matrices, typically 64-bit floating-point numbers). The short-circuting `||` behaves in exactly the same way as a boolean-valued `if`-expression.

Summing over an Array Now we can add fractional permissions to the mix: Figure 2 shows a simple, tail-recursive implementation of summing all the el-

```
let rec sum_array (!i : !int) (!n : !int) (!x0 : !elt)
                  ('x) (row : 'x arr) : 'x arr * !elt =
    if i = n then
        (row, x0)
    else
        let (row, !x1) = row[i] in
        sum_array (i + 1) n (x0 +. x1) 'x row in
    sum_array
;;
```

Fig. 2. Summing over an array in NumLin.

```
let row = row[i] := x1 in (* or *) let () = free row in
(* Could not show equality:                                          *)
(*     z arr                                                         *)
(* with                                                              *)
(*     'x arr                                                        *)
(*                                                                   *)
(* Var 'x is universally quantified                                  *)
(* Are you trying to write to/free/unshare an array you don't own? *)
(* In test/examples/sum_array.lt, at line: 7 and column: 19          *)
```

Fig. 3. Attempting to write to or free a read only array in NumLin.

ements in an array. There are many new features; first among them is `!x0 : !elt`, the type of array/matrix elements (64-bit floating point).

Second is `('x) (row: 'x arr)` which is an array with a universally-quantified fractional permission. In particular, this means the body of the function cannot mutate or free the input array, only read from it. If the programmer did try to mutate or free `row`, then they would get a helpful error message (Figure 3).

Alongside taking a `row: 'x arr`, the function also returns an array with exactly the same fractional permission as the `row` (which can only be `row`). This is necessary because of linearity: for the caller, the original array passed in as an argument would be out of scope for the rest of the expression, so it needs to be returned and then rebound to be used for the rest of the function.

An example of this consuming and re-binding is in `let (row, !x1) = row[i]`. Indexing is implemented as a primitive `get: 'x. 'x arr --o !int --o 'x arr * !elt`. Although fractional permissions can be passed around explicitly (as done in the recursive call), they can also be *automatically inferred at call sites*: `row[i] == get _ row i` takes advantage of this convenience.

One-dimensional Convolution Figure 4 extends the set of features demonstrated by the previous examples by mutating one of the input arrays. A one-dimensional convolution involves two arrays: a read-only kernel (array of weights) and an input vector. It modifies the input vector *in-place* by replacing each `write[i]` with a weighted (as per the values in the kernel) sum of it and its neighbours; intuitively, sliding a dot-product with the kernel across the vector.

```
let rec simp_oned_conv
        (!i : !int) (!n : !int) (!x0 : !elt)
        (write : z arr) ('x) (weights : 'x arr)
        : 'x arr * z arr =
    if n = i then (weights, write) else
    let !w0 <- weights[0]   in
    let !w1 <- weights[1]   in
    let !w2 <- weights[2]   in
    let !x1 <- write[i]     in
    let !x2 <- write[i + 1] in
    let written = write[i] := w0 *. x0 +. (w1 *. x1 +. w2 *. x2) in
    simp_oned_conv (i + 1) n x1 written _ weights in
  simp_oned_conv
;;
```

Fig. 4. *Simplified* one-dimensional convolution.

What's implemented in Figure 4 is a *simplified* version of this idea, so as to not distract from the features of NumLin. The simplifications are:

- the kernel has a length 3, so only the value of `write[i-1]` (prior to modification in the previous iteration) needs to be carried forward using `x0`
- `write` is assumed to have length `n+1`
- `i`'s initial value is assumed to be `1`
- `x0`'s initial value is assumed to be `write[0]`
- the first and last values of `write` are ignored.

Mutating an array is implemented similarly to indexing one: a primitive `set: z arr --o !int --o !elt --o z arr`. It consumes the original array and returns a new array with the updated value. `let written = write[i] := <exp>` is just syntactic sugar for `let written = set write i <exp>`.

Since `write: z arr` (where `z` stands for $k = 0$, representing a fractional permission of $2^{-k} = 2^{-0} = 1$), we may mutate it, but since we only need to read from `weights`, its fractional permission index can be universally-quantified. In the recursive call, we see _ being used explicitly to tell the compiler to *infer* the correct fractional permission based on the given arguments.

Squaring a Matrix *The most pertinent aspect of* NumLin *is the types of its primitives.* While the types of operations such as `get` and `set` might be borderline obvious, the types of BLAS/LAPACK routines become an *incredibly useful, automated check for using the API correctly.*

Figure 5 shows how a linearly-typed matrix squaring function may be written in NumLin. It is a *non-recursive* function declaration (the return type is inferred). Since we would like to be able to use a function like `square` more than once, it is marked with a `!` annotation (which also ensures it captures no linear values from the surrounding environment).

```
let !square ('x) (x : 'x mat) =
    let (x, (!m, !n)) = sizeM _ x in
    let (x1, x2) = shareM _ x in
    let answer <- new (m, n) [| x1 * x2 |] in
    let x = unshareM _ x1 x2 in
    (x, answer) in
    square
;;
```

Fig. 5. Linear regression (OLS): $\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$

To square a matrix, first, we extract the dimensions of the argument `x`. Then, because we need to use `x` twice (so that we can multiply it by itself) but linearity only allows one use, we use `shareM: 'x. 'x mat --o 'x s mat * 'x s mat` to split the permission `'x` (which represents 2^{-x}) into two halves (`'x s`, which represents $2^{-(x+1)}$).

Even if `x` had type `z mat`, sharing it now enforces the assumption of all BLAS/LAPACK routines that any matrix which is written to (which, in NumLin, is always of type `z mat`) does not alias any other matrix in scope. So if we did try to use one of the aliases in mutating way, the expression would not type check, and we would get an error similar to the one in Figure 3.

The line `let answer <- new (m,n) [| x1 * x2 |]` is syntactic sugar for first creating a new $m \times n$ matrix (`let answer = matrix m n`) and then storing the result of the multiplication in it (`let ((x1, x2), answer) = gemm 1. _ (x1, false) _ (x2, false) 0. answer`). `false` means the matrix should not be accessed with indices transposed.

By using some simple pattern-matching and syntactic sugar, we can:

- write normal-looking, apparently non-linear code
- use matrix expressions directly and have a call to an efficient call to a BLAS/LAPACK routine inserted with appropriate re-bindings
- retain the safety of linear types with fractional permissions by having the compiler statically enforce the aliasing and read/write rules implicitly assumed by BLAS/LAPACK routines.

Linear Regression In Figure 6, we wish to compute $\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$. To do that, first, we extract the dimensions of matrix `x`. Then, we say we would like `xy` to be a new matrix, of dimension $m \times 1$, which contains the result of $\mathbf{X^T y}$ (using syntactic sugar for `matrix` and `gemm` calls similar to that used in Figure 5, with a `^T` annotation on `x` to set `x`'s 'transpose indices'-flag to `true`).

However, the line `let x_T_x <- new (m,m) [| x^T * x |]`, works for a slightly different reason: that pattern is matched to a BLAS call to (`syrk true 1. x 0. x_T_x`), which only uses `x` once. Hence `x` can appear *twice* in the *pattern* without any calls to `share`.

After computing `x_T_x`, we need to invert it and then multiply it by `xy`. The BLAS routine `posv: z mat --o z mat --o z mat * z mat` does exactly that:

```
let !lin_reg ('x) (x : 'x mat)
             ('y) (y : 'y mat) =
  let (x, (!_n, !m)) = sizeM _ x in
  let xy <- new (m, 1) [| x^T * y |] in
  let x_T_x <- new (m, m) [| x^T * x |] in
  let (to_del, answer) = posv x_T_x xy in
  let () = freeM to_del in
  ((x, y), answer) in
lin_reg
;;
```

Fig. 6. Linear regression (OLS): $\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$

```
let !l1_norm_min (q : z mat) (u : z mat) =
  let (u, (!_n, !k)) = sizeM _ u in
  let (u, u_T) = transpose _ u in
  let (tmp_n_n , q_inv_u ) = gesv q u in
  let i = eye k in
  let to_inv <- [| i + u_T * q_inv_u |] in
  let (tmp_k_k, inv_u_T ) = gesv to_inv u_T in
  let () = freeM tmp_k_k in
  let answer <- [| 0. * tmp_n_n + q_inv_u * inv_u_T |] in
  let () = freeM q_inv_u in
  let () = freeM inv_u_T in
  answer in
l1_norm_min
;;
```

Fig. 7. L1-norm minimisation on manifolds: $\mathbf{Q^{-1}U}(\mathbf{I} + \mathbf{U^T Q^{-1} U})^{-1}\mathbf{U^T}$

assuming the first argument is symmetric, `posv` mutates its second argument to contain the desired value. Its first argument is also mutated to contain the (upper triangular) Cholesky decomposition factor of the original matrix. Since we do not need that matrix (or its memory) again, we `free` it. If we forgot to, we would get a `Variable to_del not used` error. Lastly, we return the `answer` alongside the untouched input matrices `(x,y)`.

L1-Norm Minimisation on Manifolds Figure 7 shows even more pattern-matching. Patterns of the form `let <id> <- [| beta * c + alpha * a * b |]` are also desugared to `gemm` calls. Primitives like `transpose: 'x. 'x mat --o 'x mat * z mat` and `eye: !int --o z mat` allocate new matrices; `transpose` returns the transpose of a given matrix and `eye k` evaluates to a $k \times k$ identity matrix.

We also see our first example of re-using memory for different matrices: like with `to_del` and `posv` in the previous example, we do not need the value stored in `tmp_5_5` after the call to `gesv` (a primitive similar to `posv` but for a non-symmetric first argument). However, we can re-use its memory much later to

```
      let !kalman
        ('s) (sigma : 's mat) (* n,n *)
        ('h) (h : 'h mat)     (* k,n *)
        ('m) (mu : 'm mat)    (* n,1 *)
        (r_1 : z mat)         (* k,k *)
        (data_1 : z mat)      (* k,1 *) =
        let (h, (!k, !n)) = sizeM _ h in
(*16*)  let sigma_h <- new (k, n) [| h * sym (sigma) |] in
(*17*)  let r_2 <- [| sigma_h * h^T + r_1 |] in
(*18*)  let data_2 <- [| h * mu - data_1 |] in
(*19*)  let (h, new_h) = copyM_to _ h sigma_h in
(*20*)  let new_r <- new [| r_2 |] in
(*21*)  let (chol_r, sol_h) = posv new_r new_h in
(*23*)  let (chol_r, sol_data) = potrs _ chol_r data_2 in
        let () = freeM (* k,k *) chol_r in
(*24*)  let h_sol_h <- new (n, n) [| h^T * sol_h |] in
        let () = freeM (* k,n *) sol_h in
(*25*)  let h_sol_data <- new (n, 1) [| h^T * sol_data |] in
(*26*)  let mu_copy <- new [| mu |] in
(*27*)  let new_mu <- [| sym (sigma) * h_sol_data + mu_copy |] in
        let () = freeM (* n,1 *) h_sol_data in
(*28*)  let h_sol_h_sigma <- new (n,n) [| h_sol_h * sym(sigma) |] in
(*29*)  let (sigma, sigma_copy) = copyM_to _ sigma h_sol_h in
(*30*)  let new_sigma <- [| sigma_copy - sym (sigma) * h_sol_h_sigma |] in
        let () = freeM (* n,n *) h_sol_h_sigma in
        ((sigma, (h, (mu, (r_2, sol_data)))), (new_mu, new_sigma)) in
      kalman
      ;;
```

Fig. 8. Kalman filter: see Figure 9 for the equations this code implements. Line numbers in comments refer to equivalent lines in a C implementation.

store `answer` with `let answer <- [| 0. * tmp_5_5 + q_inv_u * inv_u_T |]`. Again, thanks to linearity, the identifiers `q` and `tmp_5_5` are out of scope by the time `answer` is bound. Although during execution, all three refer to the same piece of memory, logically they represent different values throughout the computation.

Kalman Filter Figure 8 shows a NumLin implementation of a Kalman filter (equations in Figure 9). A few new features and techniques are used in this implementation:

- `sym` annotations in matrix expressions: when this is used, a call to `symm` (the equivalent of `gemm` but for symmetric matrices so that only half the operations are performed) is inserted
- `copyM_to` is used to re-use memory by *overwriting* the contents of its second argument to that of its first (erroring if dimensions do not match)

$$\mu' = \mu + \Sigma H^T (R + H\Sigma H^T)^{-1}(H\mu - \text{data})$$
$$\Sigma' = \Sigma(I - H^T (R + H\Sigma H^T)^{-1} H\Sigma)$$

Fig. 9. Kalman filter equations (credit: matthewrocklin.com).

- `let new_r <- new [| r_2 |]` creates a copy of `r_2`
- `potrs _chol_r data_2` uses a pre-computed Cholesky decomposition to multiply `data_2` $= H\mu -$ data by `r_2` $= (R + H\Sigma H^T)^{-1}$
- a lot of memory re-use; the following sets of identifiers alias each other:
  - `r_1` and `r_2`
  - `data_1`, `data_2` and `sol_data`
  - `new_h` and `sol_h`
  - `h_sol_h`, `sigma_copy` and `new_sigma`
  - `mu_copy` and `new_mu`.

With expressions as complex as those in Figure 9, it can get very difficult to ensure memory is not being re-used incorrectly/enough whilst also enusring the aliasing assumptions for the BLAS/LAPACK routine calls are guaranteed. However, using NumLin makes incorrect aliasing and memory re-use impossible. Additionally, NumLin makes it possible to spot where memory is not being re-used enough *precisely because* of its explicit allocations, aliases and frees.

3 Formal System

3.1 The Core Type Theory and Dynamic Semantics

Describe the typing rules and operational semantics here

3.2 The logical relation

Describe the step-indexed logical relation and its main properties

3.3 Soundness Theorem

State the fundamental lemma, and sketch the proof a little

4 Implementation

Talk about how you implemented NumLin and the general architecture. Talk about how simple everything is, and also about how implementing inference for fractions is.

4.1 Implementation Strategy

NumLin transpiles to OCaml and its implementation follows the structure of a typical domain-specific language (DSL) compiler. Although NumLin's current implementation is not as embedded DSL, its the general design is simple enough to adapt to being so and also to target other languages.

Alongside the transpiler, a 'Read-Check-Translate' loop, benchmarking program and a test suite are included in the artifacts accompanying this paper.

1. **Parsing**. A generated, LR(1) parser parses a text file into a syntax tree. In general, this part will vary for different languages and can also be dealt with using combinators or syntax-extensions (the EDSL approach) if the host language offers such support.
2. **Desugaring**. The syntax tree is then desugared into a smaller, more concise, abstract syntax tree. This allows for the type checker to be simpler to specify and easier to implement.
3. **Matrix Expressions** are also desugared into the abstract syntax tree through some simple pattern-matching.
4. **Type checking**. The abstract syntax tree is explicitly typed, with some inference to make writing typical programs more convenient.
5. **Code Generation**. The abstract syntax tree is translated into OCaml, with a few 'optimisations' to produce more readable code. This process is type-preserving: NumLin's type system is embedded into OCaml's (Figure 10), and so the OCaml type checker acts as a sanity check on the generated code.

A very pleasant way to use NumLin is to have the build system generate code at *compile-time* and then have the generated code be used by other modules like normal OCaml functions. This makes it possible and even easy to use NumLin alongside existing OCaml libraries; in fact, this is exactly how the benchmarking program and test-suite use code written in NumLin.

Desugaring Desugaring is conventional.

$$
\begin{aligned}
\textbf{fun}\ (x : t)\ {}'r\ (y : {}'r\ \textbf{mat}) \to e &\Rightarrow \textbf{fun}\ (x : t) \to \textbf{fun}\ {}'r \to \textbf{fun}\ (y : {}'r\ \textbf{mat}) \to e \\
x[e] &\Rightarrow \textbf{get}\ \_\ x\ (e) \qquad \text{(similarly for matrices)} \\
x[e_1] := e_2 &\Rightarrow \textbf{set}\ x\ (e_1)\ (e_2) \qquad \text{(similarly for matrices)} \\
\textbf{let}\ !x = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let Many}\ x = e_1\ \textbf{in let Many}\ x = \textbf{Many}\ (\textbf{Many}\ x)\ \textbf{in}\ e_2 \\
\textbf{let}\ f\ \langle args \rangle = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let}\ f = \textbf{fun}\ \langle args \rangle \to e_1\ \textbf{in}\ e_2 \\
\textbf{let}\ !f\ \langle args \rangle = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let Many}\ f = \textbf{Many}\ (\textbf{fun}\ \langle args \rangle \to e_1)\ \textbf{in}\ e_2 \\
\textbf{let rec}\ f\ (x : t)\ \langle args \rangle : t' = e_1\ \textbf{in}\ e_2 &\Rightarrow \textbf{let Many}\ f = \textbf{Many}\ (\textbf{fix}\ (f, x : t, \textbf{fun}\ \langle args \rangle \to e_1 : t'))\ \textbf{in}\ e_2
\end{aligned}
$$

Matrix Expressions Pattern matching!

Type Checking and Fractional Permission Inference Unification!

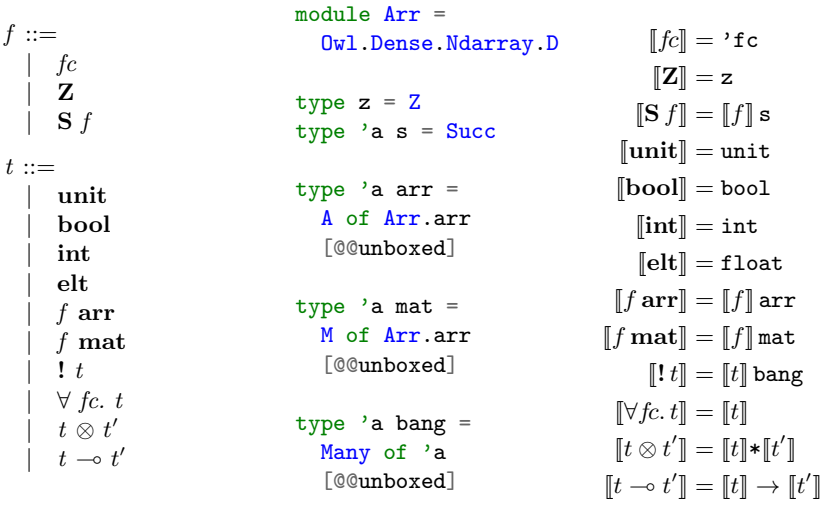

Fig. 10. NumLin's type grammar (left) and its embedding into OCaml (right).

Code Generation A few examples?

4.2 Performance Metrics

Here, evaluate the performance of the examples from the second section. Compare with your C implementations, and perhaps as well as the straightforward math transcribed into (Matlab/R/Numpy?).

5 Discussion and Related Work

The main point we want to make is that using linear types for BLAS is an "obvious" idea, but is surprisingly under-explored.

- Rust
- ATS
- Single-assignment C
- Linear Haskell
- Bernardy and Sveningsson
- L3
- Boyland fractional permissions

References